Privacy and pruning in the Mini-blockchain

B.F. França

November 2014

Abstract

In order to be useful a cryptocurrency needs to be scalable and private. Despite the dozens of cryptocurrencies launched in the last few years, none tried to solve both these problems. In this paper we improve the mini-blockchain scheme and make it private and even more scalable. We achieve this by encrypting transaction amounts and accounts balances with homomorphic encryption which allows us to perform addition and subtraction on encrypted values without revealing the plaintexts. We also introduce "expiration dates" on accounts so that users need to pay fees periodically to maintain their accounts. This way abandoned accounts are automatically pruned from the mini-blockchain.

1 Introduction

Bitcoin is an groundbreaking concept which has already made its mark into the world although it's only 5 years old. Despite all its merits, Bitcoin has major flaws that impede its adoption as a mainstream currency, namely:

Scalability: Bitcoin at this moment is only able to handle 7 transactions per second (tps), a far cry from the 10,000-100,000 tps required for it to become a global currency. Also, the blockchain grows with every transaction and even with the proposed spent output pruning it won't be satisfactorily compressed.

Speed: Bitcoin blocks are mined only at every 10 minutes so transactions only get their first confirmation, on average, after 5 minutes. Obviously, Bitcoin can't compete with existing payment systems (Visa, Paypal, etc) unless it can make reasonably secure and fast transactions.

Privacy: All transactions are public and unencrypted in the blockchain so it's very easy to follow transactions or to find the balance associated with a given address. The current suggested practices to improve privacy (use different addresses for every payment, use change addresses, use mixing services) are inefficient, increase the blockchain size and/or require third-party trust. Although Bitcoin's pseudonymity provides a reasonable

amount of privacy for individuals in their daily life, it is completely unsuitable for corporations and governments which require an higher level of privacy and for which, because of their increased transaction value and rate, the current privacy suggested practices are difficult to implement.

In the last few years there have been an explosion of cryptocurrencies and research papers trying to solve one or several of the above problems. One of them was the mini-blockchain scheme which modified the original blockchain in order to reduce its size and support increased block size. In this paper we build on the mini-blockchain scheme and modify it to achieve greater privacy and a method of "pruning" the mini-blockchain.

2 Building blocks

2.1 Mini-blockchain

The mini-blockchain (MBC) was designed by J.D. Bruce as an improvement to the original blockchain created by S. Nakamoto. The main innovation was the introduction of the "account tree", which is basically a balance sheet storing the balance of every account. With this change, transactions no need to be stored forever in the blockchain, only the most recent transactions and the current account tree need to stored. The mini-blockchain is thus much more scalable than the original blockchain since the mini-blockchain only grows when new accounts are created.

The mini-blockchain consists of 3 components:

1. Account tree
2. Transaction tree
3. Proof chain

We will now succinctly describe each one. First, the account tree is a Merkle tree of all the accounts in a given block, each account being a data block with an address and a balance (it can have more data fields, if necessary). Second, the transaction tree is a Merkle tree of all transactions in a given block, each transaction representing a change to a number of accounts. Third, the proof chain is simply a chain of blocks where each block contains a nonce, the top hash of the account tree and of the transaction for that block and the hash of the previous block. Basically, it is the headers of a normal blockchain.

The mini-blockchain operates very similarly to the regular blockchain. Every block the client nodes submit their transactions to the network. Then, each miner (independently) verify the correctness of the transactions and create a transaction tree with the correct ones. Each miner also modifies the account tree to reflect the changes done by the transactions. Finally, like in a normal blockchain, he creates the current block in the proof chain by trying different nonces until the hash of the block has a given number of zeros. If a miner

can find such a nonce, he can submit it to the network to be included in the mini-blockchain. Unlike a regular blockchain, nodes only need to keep a finite number of account trees and transaction trees so, after a new block is created, the account tree and the transaction tree of an older block is discarded. Only the proof chain needs to be stored in its entirety.

In this paper we are only going to change how accounts and transactions are coded into the mini-blockchain, everything else is going to remain intact. So, from now on, we are only going to talk about accounts and transactions, without any reference to the proof chain or the transaction tree.

2.2 Paillier cryptosystem

The Paillier cryptosystem is an asymmetric encryption algorithm based on the decisional composite residuosity assumption (DCRA). To create a key pair we first choose two primes, p and q, of equal length. The encryption key will be $N = p \cdot q$ and the decryption key will be $\lambda = (p-1) \cdot (q-1)$. To encrypt a message $m \in \mathbf{Z}_N$, we pick a random integer $r \in \mathbf{Z}_N^*$ and calculate the ciphertext as

$$Enc(m, r) = (N+1)^m \cdot r^N \ mod \ N^2$$

To decrypt, we compute the original message as

$$m = \frac{(Enc(m, r)^\lambda \ mod \ N^2) - 1}{N} \cdot \lambda^{-1} \ mod \ N$$

We can also recover the random integer r used in a given ciphertext by the formula

$$r = c^{N^{-1} \ mod \ \lambda} \ mod \ N$$

$$c = Enc(m, r) \cdot (N+1)^{-m} \ mod \ N$$

The Paillier cryptosystem is additively homomorphic, in particular it has the following properties:

$$Enc(m_1, r_1) \cdot Enc(m_2, r_2) = Enc(m_1 + m_2, r_1 \cdot r_2)$$

$$Enc(m, r)^k = Enc(k \cdot m, r^k)$$

Paillier also has the blinding property, the ability to change a cyphertext without changing the corresponding plaintext,

$$Enc(m, r_1 \cdot r_2) = Enc(m, r_1) \cdot Enc(0, r_2)$$

2.3 Elliptic curve digital signature algorithm

The elliptic curve digital signature algorithm (ECDSA) is a variant of the digital signature algorithm (DSA) using elliptic curves. To use ECDSA we first need to choose the elliptic curve parameters, namely, the curve equation and a base point P with large prime order n, meaning that $n \cdot G = \Omega$, where Ω is the "point

at infinity". These parameters can be used for several different keys if needed. To create a key pair we choose a random integer $d \in [1; n-1]$ and calculate a point $Q = d \cdot P$, the private key is going to be d and the public key is going to be Q. To sign a message m we run the following algorithm,

1. Calculate $e = H(m)$, where H is an hash function.
2. Select the $\lceil \log_2 n \rceil$ leftmost bits of e and convert them into an integer. Call the integer z.
3. Pick a random integer $k \in [1; n-1]$.
4. Calculate the point $(x_1, y_1) = k \cdot P$.
5. Calculate $r = x_1 \ mod \ n$. If $r = 0$, go back to step 3.
6. Calculate $s = k^{-1}(z + rd) \ mod \ n$. If $s = 0$, go back to step 3.
7. Return the tuple (r, s).

To verify a given signature, we run the following algorithm,

1. Calculate $e = H(m)$, where H is an hash function.
2. Select the $\lceil \log_2 n \rceil$ leftmost bits of e and convert them into an integer. Call the integer z.
3. Calculate $w = s^{-1} \ mod \ n$.
4. Calculate $u_1 = zw \ mod \ n$ and $u_2 = rw \ mod \ n$
5. Calculate the point $(x_1, y_1) = u_1 \cdot P + u_2 \cdot Q$.
6. If $r = x_1 \ mod \ n$, accept the signature.

2.4 Elliptic curve Pedersen commitment scheme

The elliptic curve Pedersen commitment scheme is a variant of the Pedersen commitment scheme based on elliptic curve cryptography. It is length-reducing so we can commit to several values using only one commitment. To create a commitment key we, of course, need a prime order elliptic curve, we are going to call the group of elliptic curve points $\mathbf{F}_p$, where p is a large prime. Then we choose m, the number of values we are going to commit to, and then pick $m + 1$ random points $P_1, ..., P_m, Q$ in $\mathbf{F}_p$. The commitment key is the tuple $(P_1, ..., P_m, Q)$. To commit to values $x_1, ..., x_m \in \mathbf{Z}_p$, we pick a random integer $r \in \mathbf{Z}_p$ and calculate the commitment as

$$Com(x_1, ..., x_m, r) = x_1 P_1 + ... + x_m P_m + rQ$$

. To open the commitment we simply reveal the values $r, x_1, ..., x_m$. EC Pedersen is also additively homomorphic, so it has the following properties:

$$Com(x_1 + y_1, ..., x_m + y_m, r_x + r_y) = Com(x_1, ..., x_m, r_x) \cdot Com(y_1, ..., y_m, r_y)$$

$$Com(k \cdot x_1, ..., k \cdot x_m, k \cdot r) = Com(x_1, ..., x_m, r)^k$$

2.5 Fiat-Shamir heuristic

The Fiat-Shamir heuristic is a method that allows one to transform an interactive proof of knowledge into a non-interactive proof of knowledge. In order to do this, the proof of knowledge must be public-coin and we must assume the random oracle model. The heuristic consists in substituting the verifier's challenges by an hash of the previous rounds of communication. As an example, the normal Schnorr's protocol for knowledge of a discrete logarithm is shown,

Statement: Prover knows a x such that $y = g^x$

Public information: y,g

Private information: x

P $\rightarrow$ **V:** Chooses random $r \in \mathbf{Z}$. Sends $t = g^r$.

V $\rightarrow$ **P:** Chooses random $c \in \mathbf{Z}$ and sends it.

P $\rightarrow$ **V:** Sends $s = c \cdot x + r$.

V: Checks if $g^s = y^c \cdot t$. If true, accepts proof.

And the corresponding non-interactive version using the Fiat-Shamir heuristic. Let $H(\cdot)$ be an ideal hash function,

Statement: Prover knows a x such that $y = g^x$

Public information: y, g

Private information: x

P $\rightarrow$ **V:** Chooses random $r \in \mathbf{Z}$. Calculates $t = g^r$.
Calculates $c = H(t)$. Calculates $s = c \cdot x + r$.
Sends the tuple (t, s).

V: Calculates $c = H(t)$. Checks if $g^s = y^c \cdot t$. If true, accepts proof.

Notice that the actual proof is only the tuple (t, s).

2.6 Linear algebra zero-knowledge arguments

J. Groth, in 2009, developed zero-knowledge arguments for a series of linear algebra relations between matrices. This was achieved in two steps. First, he used the generalized Pedersen commitment scheme to commit to matrices by creating a commitment for each row of the matrix. So, for example, if we had a matrix $\{x_{ij}\}_{i=1,j=1}^{n,m}$, we would create n commitments of the form $Com_i(x_{i1}, ..., x_{im}, r_i)$. This set of n commitments would be our committed matrix. Second, he reduced each one of the linear algebra relations mentioned in the paper to a set of simpler relations of the form:

$$z = \mathbf{x} \cdot (\mathbf{y} \circ \mathbf{h}),\ z \in \mathbf{Z}_p,\ \mathbf{x}, \mathbf{y}, \mathbf{h} \in \mathbf{Z}_p^n$$

Where $\circ$ is the Hadamard product and $\mathbf{h}$ is a vector chosen by the verifier. The zero-knowledge argument for this basic relation is the following:

Statement: $z = \mathbf{x} \cdot (\mathbf{y} \circ \mathbf{h})$

Public information: $\mathbf{h},\ a = Com(\mathbf{x}, r),\ b = Com(\mathbf{y}, s),\ c = Com(z, t)$

Private information: $\mathbf{x}, \mathbf{y}, z, r, s, t$

P $\rightarrow$ V: Choose random $\mathbf{d}_x, \mathbf{d}_y \in \mathbf{Z}_p^n$ and $d_z, r_d, s_d, t_1, t_0 \in \mathbf{Z}_p$.
Calculate and send $a_d = Com(\mathbf{d}_x, r_d),\ b_d = Com(\mathbf{d}_y, s_d),\ c_1 = Com(\mathbf{x} \cdot [\mathbf{d}_y \circ \mathbf{h}] + \mathbf{d}_x \cdot [\mathbf{y} \circ \mathbf{h}], t_1),\ c_0 = Com(\mathbf{d}_x \cdot [\mathbf{d}_y \circ \mathbf{h}], t_0)$

V $\rightarrow$ P: Choose random $e \in \mathbf{Z}$ and send it.

P $\rightarrow$ V: Send $\mathbf{f}_x = e \cdot \mathbf{x} + \mathbf{d}_x,\ \mathbf{f}_y = e \cdot \mathbf{y} + \mathbf{d}_y,\ r_x = er + r_d,\ s_y = es + s_d,\ t_z = e^2 t + et_1 + t_0$

V: Check if $ea + a_d = Com(\mathbf{f}_x, r_x)$,$eb + b_d = Com(\mathbf{f}_y, s_y)$ and $e^2 c + ec_1 + c_0 = Com(\mathbf{f}_x \cdot [\mathbf{f}_y \circ \mathbf{h}], t_z)$
If true, accept proof.

3 Non-interactive zero-knowledge proofs

Our scheme will make use of three different non-interactive zero-knowledge proofs (NIZKP). The first proof allows us to show that a Paillier encryption and a EC Pedersen commitment hide the same plaintext. It is a variant of Schnorr's protocol.

NIZKP-1

Public information: $E(x, r), Com(x, s)$

Private information: x, r, s

Construction: Pick random $u \in \mathbf{Z}_N, r_u \in \mathbf{Z}_N^*, s_u \in \mathbf{Z}_p$.
Calculate $E(u, r_u)$ and $Com(u, s_u)$.
Calculate the hash $H(E(u, r_u), Com(u, s_u))$ and transform the output into an integer $z \in \mathbf{Z}_p^*$.
Calculate $w = z \cdot x + u$, $r' = r^z \cdot r_u$ and $s' = z \cdot s + s_u$.

Proof: $E(u, r_u), Com(u, s_u), w, r', s'$

Verification: Calculate $E(w, r')$ and $Com(w, s')$.
Calculate the hash $H(E(u, r_u), Com(u, s_u))$ and transform the output into z.
Check if $E(w, r') = E(x, r)^z \cdot E(u, r_u)$ and $Com(w, s') = Com(x, s) \cdot z + Com(u, s_u)$.
Accept if both are true.

Another version of the above proof works with a vector of values instead of a single value.

NIZKP-2

Public information: $E(x_1, r_1), ..., E(x_n, r_n), Com(x_1, ..., x_n, s)$

Private information: $x_1, ..., x_n, r_1, ..., r_n, s$

Construction: Pick random $u_1, ..., u_n \in \mathbf{Z}_N; r_{u1}, ..., r_{un} \in \mathbf{Z}_N^*; s_u \in \mathbf{Z}_p$.
Calculate $E(u_1, r_{u1}), ..., E(u_n, r_{un}), Com(u_1, ..., u_n, s_u)$.
Calculate the hash $H(E(u_1, r_{u1}), ..., E(u_n, r_{un}), Com(u_1, ..., u_n, s_u))$ and transform the output into an integer $z \in \mathbf{Z}_p^*$.
Calculate $w_1 = z \cdot x_1 + u_1, ..., w_n = z \cdot x_n + u_n, r'_1 = r_1^z \cdot r_{u1}, ..., r'_n = r_n^z \cdot r_{un}, s' = z \cdot s + s_u$.

Proof: $E(u_1, r_{u1}), ..., E(u_n, r_{un}), Com(u_1, ..., u_n, s_u), w_1, ..., w_n, r'_1, ..., r'_n, s'$

Verification: Calculate $E(w_1, r'_1), ..., E(w_n, r'_n), Com(w_1, ..., w_n, s')$.
Calculate the hash $H(E(u_1, r_{u1}), ..., E(u_n, r_{un}), Com(u_1, ..., u_n, s_u))$ and transform the output into z.
Check if $\forall i \in \{1, ..., n\}$, $E(w_i, r'_i) = E(x_i, r_i)^z \cdot E(u_i, r_{ui})$ and $Com(w_1, ..., w_n, s') = Com(x_1, ..., x_n, s) \cdot z + Com(u_1, ..., u_n, s_u)$.
Accept if both are true.

Note that the above proof works even if each one of the encryptions is done with a different key.

The last proof proves that, for the committed value y and the committed vector $\mathbf{x} = (x_1, x_2, ..., x_n)$, the following relation holds $y = x_1 + x_2 + ... + x_n$.

NIZKP-3

Public information: $Com(\mathbf{x}, r), Com(y, s)$

Private information: $\mathbf{x}, r, y, s$

Construction: Create the vector $\mathbf{1} = (1, 1, ..., 1)$.
Pick random $\mathbf{d} \in \mathbf{Z}_p^n$ and $r_a, s_1, s_2 \in \mathbf{Z}_p$.
Calculate the commitments $a = Com(\mathbf{d}, r_a), c_1 = Com(\mathbf{x} \cdot \mathbf{1} + \mathbf{1} \cdot \mathbf{d}, s_1), c_2 = Com(\mathbf{d} \cdot \mathbf{1}, s_2)$.
Calculate the hash $H(a, c_1, c_2)$ and transform the output into an integer $e \in \mathbf{Z}_p^*$.
Calculate $\mathbf{f} = e\mathbf{x} + \mathbf{d}, r'_a = er + r_a, t = e^2 s + e s_1 + s_2$.

Proof: $a, c_1, c_2, \mathbf{f}, r'_a, t$

Verification: Create the vector $\mathbf{1} = (1, 1, ..., 1)$.
Calculate the hash $H(a, c_1, c_2)$ and transform the output into e.
Calculate $\mathbf{g} = e\mathbf{1} + \mathbf{1}$.
Check if $Com(\mathbf{x}, r)e + a = Com(\mathbf{f}, r'_a)$ and $Com(y, s)e^2 + c_1 e + c_2 = Com(\mathbf{f} \cdot \mathbf{g}, t)$.
Accept if both are true.

4 Description

As we have said previously, from a conceptual point of view we only have accounts and transactions. First, we will describe an account. An account is a data block with the following data fields,

Address: an alphanumeric string chosen by the account owner.

Account key: a Paillier encryption key.

Balance: the balance of the account, encrypted using the account key.

Expiration block: the number of the block in which the account expires.

Signature threshold: the number of signatures required to make transactions.

Signature keys: ECDSA public keys, there may be several.

Accounts also have some private information associated with it, namely the corresponding Paillier decryption key and ECDSA private keys. Now we will describe transactions. A transaction is any request made to the network to change the account tree in any way. There are only three transactions,

`create`**:** creates a new account.

`update`**:** changes the expiration block, the signature threshold or the signature keys associated with an account.

`transfer`**:** transfer funds from one account to other account(s).

In our scheme an account is created by a `create` transaction, no transfer of coins is needed. This way we can choose the address of our account (and make it easy to remember) and we have a simple way to create multi-signature accounts (we only need to insert the extra signatures on the `create` transaction). When an account is created it is created with a pre-determined "lifetime", this "lifetime" is the number of blocks until the account expires and is equal to the expiration block number minus the current block number. Accounts are always created with the same lifetime and this lifetime is hard-coded into the protocol.

After an account is created it can receive and send coins like a normal account and also be updated. We can update our account with an `update` transaction, this allows us to change the signature keys or the signature threshold and, most importantly, increase the expiration block. If we don't want our account to expire, we need to send periodic `update` transactions to increase the expiration block. Of course, if we want we can increase the expiration block by a big amount, so that the account only expires in one year or more. This type of "expiration date" for accounts has two very beneficial effects for the network. First, it gives an economic incentive for miners to store the account tree because they receive transactions fees on each `update` transaction. Second, it deletes abandoned accounts thus keeping the account tree small.

The balance of our account is always kept encrypted using the account key. No one can discover what our balance is without the private key. However, we can still receive and send coins because of the homomorphic properties of Paillier encryption. We can encrypt the amount of the transfer and the network can subtract or sum it to our balance without knowing either the amount or the balance. This also means that when we do a transfer, no one except the receiver know the amount of the transfer and that we don't know the receiver's balance and vice-versa. The only information that is public is that the transfer occurred and who the sender and the receiver were.

We will now describe in more detail each of the transactions.

4.1 `create` transaction

To create an account Alice first chooses an address A and checks, by looking in the account tree, that there is no account with the same address. Then she creates a Paillier key pair by choosing two random primes p, q of equal length. The public key will be $N = pq$ and the private key will be $\lambda = (p-1) \cdot (q-1)$. She also chooses the signature threshold $T_{sig} \in [1, n_{sig}]$, where n_{sig} is the total number of signatures. Finally, she creates the n_{sig} ECDSA key pairs. For each pair, the private key is a random integer d and the public key is the point $Q = d \cdot P$. Alice then sends to the network the following data,

- Address, A
- Account key, N
- Signature threshold, T_{sig}
- Signature keys, $Q_1, ..., Q_n$

She also saves the private keys λ and $d_1, ..., d_n$. A miner first checks that there are no accounts with the same address. He then includes the transaction in the next block and adds to the account tree an account with the values requested in the transaction, expiration block $B_{exp} = B_{cur} + L_{init}$ (where B_{cur} is the current block number and L_{init} is the standard initial lifetime) and encrypted balance $E(0, 1) = 1$ (since for all N, $E(m = 0, r = 1) = (N+1)^0 \cdot r^N = 1$).

Why should the network accept a `create` transaction if there isn't any monetary reward? The answer is that there is an *indirect* monetary reward, assuming the transaction is honest. To understand why, imagine that Carol creates an account and receives some coins at that account. Now she has three options:

1. Extend the lifetime of her account. To do this she will have to pay transaction fees.
2. Transfer the coins to another account. To do this she will have to pay transaction fees.
3. Let the account expire. She won't pay transaction fees but she will lose the balance of the account.

So, if an honest node wants to create an account, it will eventually pay transaction fees.

4.2 `update` transaction

Once Alice has an account, she can change some values using the `update` transaction. The `update` transaction consists of at least one the following data fields,

- New expiration block, B'_{exp}
- New signature threshold, T'_{sig}
- Signature keys to add, $Q'_1, ..., Q'_s$
- Signature keys to remove, $Q_1, ..., Q_r$

And of all the following data fields,

- Address, A
- Transaction fee, t
- Balance after transaction, $E(b', r'_b)$
- Zero-knowledge proof of balance
- Signatures

The zero-knowledge proof of balance guarantees that Alice has enough coins in her account to pay the transaction fee. For producing the zero-knowledge proof of balance we are going to assume two things. First, that the coin supply is $2^\alpha - 1,\ \alpha \in \mathbb{N}$. Second, that someone created a Pedersen commitment key of the form $(p, h, g_1, ..., g_\alpha);\ h, g_1, ..., g_\alpha \in \mathbf{Z}_p^*$, where p is a large prime. This key can be generated by the developers and hard-coded into the protocol.

Note that Alice has already calculated and encrypted the balance of her account after the transaction, $b' = b - t$. To create the proof of balance Alice first downloads the balance of her account, $E(b, r_b)$. If Alice doesn't know r_b she can calculate it from $E(b, r_b)$ and the private account key (see section 2.2). What Alice wants to prove is that $b = b' + t$, thus proving that her account has sufficient coins to pay the transaction fee. Then Alice commits to b, $Com(b, s_b)$, and uses NIZKP-1 to prove that it hides the same plaintext as $E(b, r_b)$. After, Alice commits to the tuple (b', t), $Com(b', t, s)$ and uses NIZKP-2 to prove that it hides the same plaintext as $E(b', r'_b)$ plus the transaction fee t. Lastly, Alice uses NIZKP-3 on the commitments $Com(b, s_b)$ and $Com(b', t, s)$ to show that $b = b' + t$. The entire zero-knowledge proof of balance is thus $Com(b, s_b)$, $Com(b', t, s)$, NIZKP-1, NIZKP-2 and NIZKP-3.

Lastly, Alice needs to sign the transaction. Note that the number of signatures required is equal to the current signature threshold.

The miner who receives the transaction verifies that the zero-knowledge proof of balance and the signatures are valid and, if he decides that the transaction fee is enough, includes the transaction in the next block. Then he updates the requested fields in the account and changes the account balance to $E(b', r'_b)$.

4.3 `transfer` transaction

Now Alice wants to transfer some coins to n different addresses, for this she will use a `transfer` transaction. Suppose she wants to transfer x_1 coins to address A_1, x_2 coins to address A_2,...,x_n coins to address A_n and that the transaction fee is t. The `transfer` transaction consists of the following fields,

- Address, A
- Output addresses, $A_1, ..., A_n$
- Output amounts, $E_1(x_1, r_1), ..., E_n(x_n, r_n)$
- Transaction fee, t
- Balance after transaction, $E(b', r'_b)$
- Zero-knowledge proof of balance
- Signatures

Note that all the amounts, except the transaction fee, are encrypted. To do this, Alice downloads the account keys of all the n addresses to which she wants to send coins. Then she encrypts each x_i using the account key of the corresponding output address A_i, so she gets $E_i(x_i, r_i)$, and also encrypts b' using her account key, $E(y, r_y)$.

The zero-knowledge proof of balance proves that Alice has enough balance to pay the transaction by showing that $b = b' + x_1 + ... + x_n + t$. To create the proof of correctness Alice downloads the balance of her account, $E(b, r_b)$, then she commits to each of the values $b, b', x_1, ..., x_n, t$ by calculating the commitments $Com(b, s_b)$ and $Com(b', x_1, .., x_n, t, s)$. Alice then uses NIZKP-1 to prove that $Com(b, s_b)$ hides the same plaintext as $E(b, r_b)$. After, she will use NIZKP-2 on $Com(b', x_1, .., x_n, t, s)$ to prove that it hides the same plaintexts as the encryptions $E(b', r'_b), E_1(x_1, r_1), ..., E_n(x_n, r_n)$ plus the transaction fee t. Then Alice will use NIZKP-3 on $Com(b', s_b)$ and $Com(b', x_1, .., x_n, t, s)$ to prove that $b = b' + x_1 + ... + x_n + t$. So, the entire zero-knowledge proof of balance is the commitments $Com(b, s_b)$ and $Com(b', x_1, .., x_n, t, s)$ and the proofs NIZKP-1, NIZKP-2 and NIZKP-3.

Lastly, Alice signs the transaction with the required number of signatures.

When a miner receives a `transfer` transaction it verifies that the proof of balance and the signatures are valid. Then it updates the balance of Alice's acount to $E(b', r'_b)$, and adds the output amounts to the corresponding output accounts by changing the corresponding balances to $E_i(b_i + x_i, q_i \cdot r_i) = E_i(b_i, q_i) \cdot E_i(x_i, r_i)$. Despite having processed the transaction the miner has only learned the output addresses, he doesn't know the amounts in the transaction or the balances of the accounts. Alice has also not learned the balances of any of the output addresses.

5 Anonymity

So far our protocol gives the users more privacy than Bitcoin since all account balances and transfer amounts are hidden but, the addresses of the sender and the receivers of a transfer are still public information so, the protocol isn't anonymous. An object is said to be anonymous, in a mathematical sense, if it is indistinguishable from all other objects in a larger set (called the "anonymity set"). For example, the traffic of a Tor user is indistinguishable from the traffic of all other Tor users but it is distinguishable from the traffic of a normal Internet user so, Tor users are anonymous in the set of all Tor users. In a cryptocurrency, ideally, the senders and the receivers of a transfer are indistinguishable from all other users of the cryptocurrency. This what we call "maximal anonymity" and so far, in the world of cryptocurrencies, has only been achieved by the Zerocoin and Zerocash protocols, both of which have serious barriers to implementation. A weaker form of anonymity, which we call "partial anonymity", has been accomplished by mixing protocols (Bitcoin Fog, Coinshuffle, Darkcoin, etc) and the Cryptonote protocol. With partial anonymity the senders and/or the receivers of a transfer are indistinguishable from a small subset of cryptocurrency users. Although it is not ideal, partial anonymity is enough for the vast majority of cryptocurrency users.

Our protocol also supports partial anonymity of the receivers of a transfer by use of the blinding property of the Paillier cryptosystem. Imagine Alice wants to send some coins to Bob but doesn't want anyone to know *with certainty* that the coins went to Bob. To do this she chooses the number of accounts that will be in the anonymity set, suppose she chose n. Then she picks $n-1$ random addresses from the account tree. Finally, she executes a transfer in which she sends the desired number of coins to Bob's address and zero coins to all other $n-1$ random addresses. Now, because of the blinding property of Paillier encryption, the cyphertexts of the balances of the $n-1$ random accounts that Alice chose will change but the actual balances won't.

$$Enc(m, r_1 \cdot r_2) = Enc(m, r_1) \cdot Enc(0, r_2)$$

An attacker who sees the transfer won't be able to know if Alice actually transferred coins to a given address or not. In fact, only Alice knows to which addresses she transferred coins. Even Bob only knows that he received some coins, he knows nothing about the amounts transferred to the other address. So, an attacker who knows that Alice transferred coins to only one account out of the n accounts in the transfer, has only $1/n$ probability of choosing Bob's account.

However, if Alice intends to make several transfers to Bob, she will need to always use the same anonymity set of accounts that she used in the first transfer. Otherwise, an attacker will see several transfers where Bob's address appears together with other addresses that never appear again (or appear less often) and will assume that Alice made those transfers to Bob's address while trying to hide it by using different anonymity sets for each transfer.

6 Conclusion

We have introduced an improvement to the original mini-blockchain scheme that supports custom addresses, automatic account tree pruning and increased privacy. This was done through the introduction of limited lifetimes for accounts and homomorphic encryption. Unfortunately, in our scheme, the transaction data size is increased relatively to the original blockchain and Bitcoin because of the use of zero-knowledge proofs. However, this increase is probably unavoidable if we want cryptographically secure privacy in a cryptocurrency.